Scalable and Resilient Workflow Executions on Production Distributed Computing Infrastructures

Javier Rojas Balderrama, Tram Truong Huu, Johan Montagnat
I3S Laboratory, MODALIS Team
University of Nice–Sophia Antipolis / CNRS UMR 7271, France
Email: javier@i3s.unice.fr, tram@polytech.unice.fr, johan@i3s.unice.fr

***Abstract*—In spite of the growing interest for grids and cloud infrastructures among scientific communities and the availability of such facilities at large–scale, achieving high performance in production environments remains challenging due to at least four factors: the low reliability of very large–scale distributed computing infrastructures, the performance overhead induced by shared facilities, the difficulty to obtain fair balance of all user jobs in such an heterogeneous environment, and the complexity of large–scale distributed applications deployment. All together, these difficulties make infrastructure exploitation complex, and often limited to experts. This paper introduces a pragmatic solution to tackle these four issues based on a service-oriented methodology, the reuse of existing middleware services, and the joint exploitation of local and distributed computing resources. Emphasis is put on the integrated environment ease of use. Results on an actual neuroscience application show the impact of the environment setup in terms of reliability and performance. Recommendations and best practices are derived from this experiment.**

***Keywords*-Grid Computing, Service Oriented Architecture, Scientific Workflow, Distributed Computing Infrastructure**

I. INTRODUCTION

Distributed Computing Infrastructures (DCIs) are being increasingly exploited for tackling the computation needs of large–scale applications. DCI middleware helps users in exploiting seamlessly large amounts of computing resources. However, executing large–scale applications on a DCI faces several well–identified problems often causing poor application performance, either under–performing execution time or complete application failure. In particular, the following show-stoppers are recurrently reported in the literature dealing with large–scale distributed applications enactment:

- Low reliability of the infrastructure causing high failure rates and severe performance losses.
- High latency of computing tasks submitted to production batch systems causing low performance.
- Unfair balance between shorter and longer computation tasks.
- Complex deployment of distributed computing applications.

Users often face critical decisions that require expert skills to setup and tune a workable solution for their particular application among available middleware services. In this paper, we propose a comprehensive and integrated execution environment designed to tackle simultaneously the low reliability, high latency, unfair balance, and complex deployment issues. The designed software architecture is reusing as much as possible existing stable middleware services. Their integration is far from trivial as different services may be tackling different issues but they are not necessarily compatible nor interoperable. The resulting framework does not only deliver to users a uniform interface to manipulate and execute their applications. It also shields users from the technical issues of the underlying infrastructure. It is exploited by neuroscientists to study neurodegenerative diseases such as Alzheimer's. The users' main concern is completely non-technical in this area.

This proposition assumes that the applications to execute are described as scientific workflows. Workflows formalize the description of distributed computing processes and they have been widely adopted among computational scientific communities. Workflows are decomposed into many application services with inter–dependencies which define ordering constraints at execution time. In scientific workflows, input datasets are usually composed of many independent data items to be processed, hence implying a high level of data parallelism. The workflow services are invoked multiple times, to process all data segments received. An application can then be seen as an orchestration of collaborating services deployed within a Service–oriented Architecture (SOA).

Our methodology is heavily based on SOA principles for their high versatility. As shown in section III, the provision of both middleware and application components as services makes it possible to fine tune application deployment and configuration, thus tackling the problems of reliability, fair balancing, and application deployment. Furthermore, high-latency concerns are addressed by advanced tasks management services. To take advantage of the service–oriented principles while being non–intrusive for the existing application services, our implementation includes an intermediate layer between the infrastructure stack of technologies and the user interfaces.

The rest of the paper is structured as follows. First, a requirements analysis is conducted and relevant technical solutions are reviewed in section II. Then, section III describes a model to decide computation tasks dispatching integrating local resources to DCIs. Section IV presents the execution framework and describes all its software components. This section also highlights optimization methods to address the failure recovery, resource reservation, and scalability of the framework. An experimental validation is presented in section V. Finally conclusion and some recommendations derived from the experiments are developed in section VI.

II. Addressing production DCIs shortcomings

A lot of research efforts have been invested in dealing more or less independently with the well–known issues of large–scale infrastructures outlined above [1], [2], [3], [4]. The goal of this work is to design an end–to–end execution framework simultaneously tackling the most critical showstoppers commonly encountered. Specifically, four issues are addressed to improve the application performance on production DCIs within the framework:

1) **Failure recovery**. Networking and computing infrastructures are subject to random resource failures. The likeliness of failures increases with the number of physical entities, as seen in large–scale distributed systems today [5], [6]. Recovering from failures becomes a critical issue to improve the reliability of the infrastructure, preventing the correct completion of many application runs. Numerous works addressing this issue have been proposed in the literature including check–pointing, live migration [7], [8], job replications [2] and submission strategies [4]. On general purpose production infrastructures, job resubmission is often the only general failure recovery solution available, as check–pointing and migration usually either make restrictive assumptions on the computational processes or they require application instrumentation. The final makespan could be increased, specifically with longer applications, but resubmission ensures that the application execution can always continue and finish successfully. This approach is implemented in the framework by controlling the status of submitted jobs and defining a resubmission policy when a failure occurs.
2) **Lowering latency**. The splitting of an application's computation logic in many tasks lends towards more parallelism but the gain may be easily compensated by the time needed to handle all tasks generated in a competitive production batch system. In the case of a workflow–based application with inter–dependencies between tasks, the sequential submission of tasks to long batch queues will be highly penalizing. Addressing the high latency issue, many works study multiple submissions approaches [1], [2], [4]. The results of these studies confirm that submitting tasks several times increases application performance. However, users who do not use multiple submission are penalized. Furthermore, without considering the capacity of batch schedulers, high number of submissions can overload the batch schedulers and then degrade the overall system performance.
   Alternatively, pilot jobs systems help users in reserving a pool of computing resources during the execution of the application [9], being considered as a bridge between batch systems and systems supporting resources reservation. A pilot job is submitted to a workload manager to reserve a computing resource. User jobs are then pulled from the job queue to computing nodes by successfully started pilot jobs. Each pilot job can thus process sequentially several user jobs without introducing delay between two of them. Each pilot is subject once to the workload manager queuing time but the jobs they process are not. Another advantage of pilot jobs to the classical submission approach include the sanity checks of the running environment before assigning resources for execution. They also allow users to create a virtual private network of computing resources reserved for executing their tasks, and they implement effectively the pull scheduling paradigm. Our execution framework extensively uses pilot jobs reducing latency and making executions more reliable because broken resources are filtered by the pilot jobs.
3) **Task fairness**. The very complex tuning of large–scale submission systems, involving meta–brokers and many schedulers, makes it extremely difficult to achieve fair balance between short and long tasks in a computation process. Yet, production infrastructures are not only used for long running jobs processing data–intensive applications but they are also frequently used for processing shorter jobs. Statistical results shows that more than 50% of the jobs take less than 30 minutes for execution [3]. While the high latency has less impact on long running jobs, short jobs are heavily penalized if they have long waiting times before execution. The larger the computing time discrepancy between tasks, the higher the impact. Users therefore require a mechanism of resource fair sharing to avoid that long jobs monopolize the whole computing resources, and delay the completion of other users' (short) jobs.
   Pilot jobs also improve handling of short jobs as they reduce individual jobs queuing time. However, although dedicated to a specific user, pilot job systems usually do not implement fairness among the user's jobs and pilots may be overloaded by the processing of longer jobs similarly to a Grid meta–scheduler. Therefore, our approach combines more dedicated resources out of a distributed infrastructure with the capacity of DCIs to improve handling of short jobs. Local resources are more reliable since the user is administrator of computing nodes, thus failures coming from the software dependencies are lowered. Executing applications locally reduces the number of job submissions remotely removing the submission phase and delays of middleware initialization. This then reduces the waiting time of other jobs in the queue for obtaining computing resources on remote infrastructures. Nevertheless, as the number of computing resources in the local server is limited, the more jobs submitted locally, the longer is the execution time needed to finish all jobs. We define a decision model in section III to decide whether a task is executed on local resources or submitted to a DCI.
4) **Deployment & scalability**. Beyond middleware parameterization, the deployment of application services may have a strong impact on application performance as servers easily become overloaded in large–scale runs [10]. Some initiatives like GASW [11] or LONI Pipeline [12] propose tools to reuse scientific applications on DCIs but they have scalability limitations or interoperability constraints respectively. Concerning Web

service–related projects, tools such as GEMLCA [13], and gRAVI [14] manage services lifecycle at different levels, enabling dynamic deployment and/or supporting of non–functional concerns. However, their adoption involves the use of an homogeneous middleware. Our execution framework relies on a legacy application code wrapper that both provides a standard Web service interface to all application computing components, and helps managing the complete lifecycle of the resulting services.
In practice multiple services containers, acting as a proxy between users and the production DCI, may be configured in the framework, as described in section IV. Each container naturally has a limited capacity to process concurrent services. When the size of input dataset increases, the number of services submitted concurrently may exceed its capacity. The replication of servers into the system (scaling out) resolves this limitation. It increases the performance without modifying the framework architecture.

The execution framework described below addresses simultaneously the production DCIs shortcomings by combining advanced job submission strategies, services replication, and including the use of local resources during workflow enactment. The implementation of job resubmission improves the reliability by instantiating a system capable of error overcoming from remote executions. Then the adoption of pilot jobs for multilevel scheduling ensures the reduction of latency. Pilot jobs represent a new approach to overcome long queues of batch schedulers reusing computing resources efficiently. In order to tackle the unfair balancing resulting from the competition of short/lightweight application tasks with the long/heavyweight ones, a decision model dispatching tasks among local and remote resources is implemented. The deployment of services provides transparent mechanisms of applications reallocation, over local and remote resources, holding back technical details far from final users. Finally, the scalability heedfulness ensures large–scale experiment campaigns by enabling services resiliency.

The delivery of an integrated execution environment is eased by the application of SOA principles, made possible by the workflow formalism used to model distributed applications. SOA has been adopted to a large extent in middleware design [15]. For instance, the Swift workflow management system [16] provides an integrated working environment for job scheduling, data transfer, and job submission. It is built on top of a uniform implementation based on Globus toolkit. Yet, production infrastructures hardly ever comply to a homogeneous middleware stack, nor adopt a single communication standard for all core and community services. Conversely, traditional workflow management systems like Taverna [17], or Triana [18] support service invocation enabling interoperability but they do not natively execute code on DCIs. In our architecture, both middleware and application components are deployed as services. The application code is instrumented non–invasively to comply to this model through a Web service builder aware of DCIs computing capability [19]. Using an SOA approach allows users to scale the execution of their applications and flexibly extend the execution framework according to the computation needs.

III. Model for efficient use of local resources

In spite of the large number of computing resources available on DCIs, the waiting time of a job to obtain a computing resource may increase considerably with a big number of jobs simultaneously submitted to the infrastructure. This latency is particularly not negligible for short–execution jobs. Using local resources may then complement DCIs resources. By reducing the number of short jobs executed remotely, they reduce the management time of jobs processed on the DCI thus improving the reliability and performance of the application. However, a strategy is required to ensure that local resources are not overloaded when many jobs are executed. In view of this, a decision model is defined to dispatch incoming jobs for local execution or for submission to a DCI based on expertises captured from the application. It makes the assumption that each workflow activity i among the k used activities $\{i \in \mathbb{Z}^+ \mid i \leqslant k\}$ is consuming a fixed amount of resources when executing (i.e., r_i memory space, and t_i execution time). It is also assumed that the target DCIs are large enough to handle simultaneously all computation tasks triggered by the invocation of the application services at runtime.

Let R denote the memory consumed on the local resource for all running services including r_j which would be an incoming service of type j executed locally at a given time. The value of R is computed according to Equation 1, where n_i denotes the number of services of type i. The volume of assigned memory must not exceed R_{MAX}, the available memory installed on the local resource ($R \leqslant R_{\text{MAX}}$).

$$R = r_j + \sum_{i=1}^{k} n_i \times r_i \tag{1}$$

Making the hypothesis that production infrastructures have sufficient computing resources to execute all submitted services, the execution time of a scientific workflow T_{MAX} would be the longest path of its representation as a graph (aka critical path). Therefore, the execution time in the local resources T must be shorter than this theoretical threshold in order to avoid penalizing the final execution time of the workflow ($T \leqslant T_{\text{MAX}}$). The value of T, as shown in Equation 2, represents the sequential execution time of all services running on local resources distributed on all available processor units, where N_{CPU} denotes the number CPU cores.

$$T = \frac{t_j + \sum_{i=1}^{k} n_i \times t_i}{N_{\text{CPU}}} \tag{2}$$

Algorithm 1 shows the procedure to decide whether a job is executed locally or submitted remotely. The estimation of R and T is performed each time an incoming service is enacted by the workflow manager. Meanwhile, the value of n_j is updated for accounting.

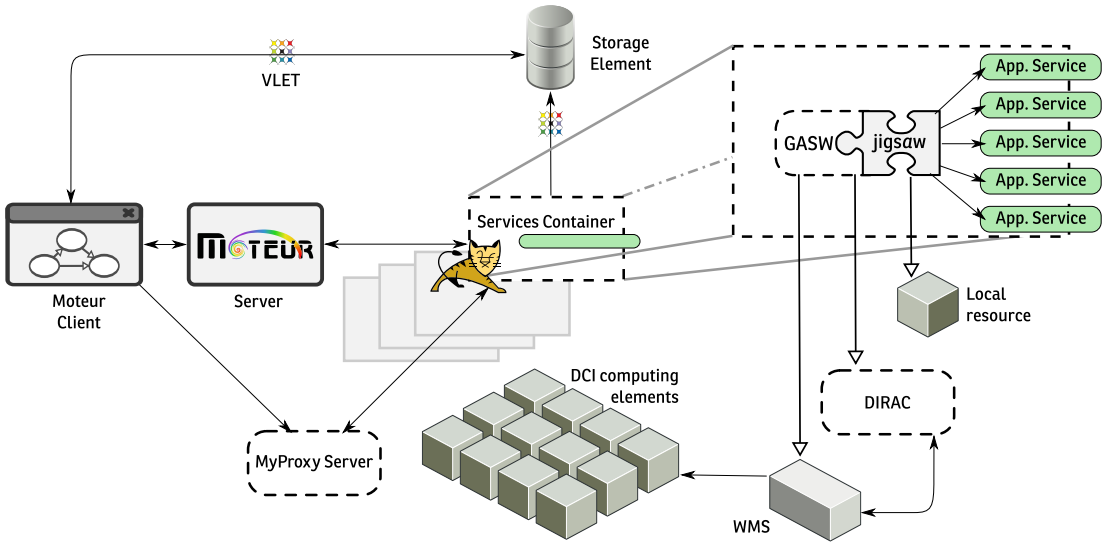


Fig. 1: Overall framework architecture

Algorithm 1 Dispatching of incoming service (r_j, t_j)

Require: $[r_i]_k$ memory benchmark of services
Require: $[t_i]_k$ execution time benchmark of services
Require: $[n_i]_k$ number of services running by type
Require: R_{MAX}, T_{MAX}, and N_{CPU}

for i in $\{1, \ldots, k\}$ **do**
  $R = R + r_i \times n_i$
  $T = T + t_i \times n_i$
end for
$R = R + r_j$
$T = (T + t_j)/N_{\text{CPU}}$
if $R \leqslant R_{\text{MAX}}$ **and** $T \leqslant T_{\text{MAX}}$ **then**
  service is executed locally
  $n_j = n_j + 1$
else
  service is submitted to a DCI
end if

IV. Execution framework

This section describes a highly configurable, and standards-based service architecture enabling reproducible and scalable experimentation. From a user's point of view, an end–to–end system enabling the exploitation of DCIs should provide high expressiveness to describe applications; design and enact applications composition making use of consistent interfaces; and transparent access to DCI resources. The framework architecture, based on MOTEUR [20] and *jigsaw* [19] components, is pictured in Fig. 1.

MOTEUR, a scientific workflow environment, is the front–end component that connects the user to the rest of the framework. It includes a workflow designer and an execution monitor. A MOTEUR client interacts with the MOTEUR server at runtime to execute the application considering a specific input dataset. The MOTEUR server is responsible for invoking each application service through generic interfaces. Before each invocation, it executes the decision model algorithm to define whether the job should be executed locally or submitted remotely addressing the task fairness described in the third DCI shortcoming of section II. In our architecture, application services invoked by MOTEUR are packaged and instrumented using the *jigsaw* framework. *Jigsaw* is a legacy application code wrapper which exposes legacy application components as standard Web services, and provides a complete mechanism to handle the invocation of application code locally, on the server hosting the service, or remotely, on the DCI.

Jigsaw enables the definition of multiple execution profiles corresponding to different execution platforms, and it is responsible for monitoring the status jobs and it implements resubmission policies when an execution fails resolving the first DCI shortcoming concerning failure recovery. It integrates multiple submission back–ends such as GASW [11], to target various execution infrastructures. It can dispatch executions on the local server (local resource), PBS clusters (not represented here) or the European Grid Infrastructure (EGI) through its Workload Management System interface (WMS). In addition, the DIRAC pilot management system [9] is integrated in the framework ahead of the default WMS and its computing resources. The DIRAC architecture consists of numerous co-operating distributed services and light agents built within the same framework following distributing security standards. It implements a multilevel scheduling using pilot jobs that drastically improves reliability of jobs submission and significantly reduces task latency. This addresses the second shortcoming, and overcomes the full list of identified strategies to improve the applications performance on production DCI environments.

Jigsaw instruments the legacy code execution interfaces using WS–compliant messages for services invocation. The execution description, packaged along with the legacy codes in portable artifacts, are deployed and published in a services container such as Apache Tomcat [21], thus tackling the complete services lifecycle, from creation to deployment and then invocation. The *jigsaw* services may be hosted and replicated

in several containers. It makes possible to scale the system out enforcing the availability of services, and distributing the workload associated to the invocation of services involved in task dispatching. Together, *jigsaw* and the services container, provide an integrated solution to the deployment and scalability arguments mentioned in the fourth shortcoming of the requirements analysis.

Complementary modules are integrated in the framework to address non–functional requirements of users' authentication and data management. User's credentials may be required for authentication during execution. Therefore, each framework component can connect to a MyProxy server [22]. The user is just required to provide the login and password of the credential stored on the MyProxy server. The validity of the proxy is checked each time an operation is performed. An expired proxy will be automatically renewed without interrupting the application execution. Concerning data transfers, the VL-e Toolkit [23] is also integrated in the framework. It provides a unified view of heterogeneous file systems. VL-e Toolkit supports several protocols such as gridFTP or HTTP, and file schemes like the grid LFN or local file systems. It is used for service deployment, and file staging on the services container to provide the data inputs to the service instances enforcing the interoperability.

The framework targets a coherent integration of a data-driven approach to manipulate complex data structures through high level interfaces. The MOTEUR client provides to users a graphical application to configure services and describe the semantics of dataflows, achieving transparent parallelism. The description is represented by the GWENDIA workflow language [24], that supports the required expressiveness to represent services composition. While the MOTEUR client offers design tools to build workflows and configure an execution environment, the MOTEUR server provides asynchronous invocation and orchestration of services, for optimizing the execution of data–intensive workflows. The integration of *jigsaw* with MOTEUR provides a full range of functionality, making them suitable for the purpose of applications reuse and large–scale experimentation.

V. Results

Experiments have been designed to validate the submission model and the execution framework. The framework is stress tested using a real application related to the progression of Alzheimer's disease. The experimental setup aims at quantifying the endured by, and the speedup of the application.

A. Case study

Neuro–degenerative diseases like Alzheimer's disease are characterized by a co–occurrence of different pathological phenomenas which eventually cause brain cells loss over time. Monitoring the structural changes of the brain provides a way to track the evolution of the disease. The evaluation of changes in time from serial data of the same subject acting as his own control (i.e., longitudinal analysis) is useful for detecting the subtle changes related to the biological processes. This original processing was developed at the Asclepios Research Project from Inria Sophia–Antipolis [25]. Its scientific workflow representation is shown in Fig. 2. It includes pre–treatment steps for reorientation, and registration of images against an atlas of the brain. Then the intensity of images is normalized, and a non–rigid registration is performed to look for anatomical differences between pairs of images. Finally, a quantification of the longitudinal brain atrophy is derived from the Jacobian matrix of deformations as an average measuring the volume change in comparison to a reference mask.

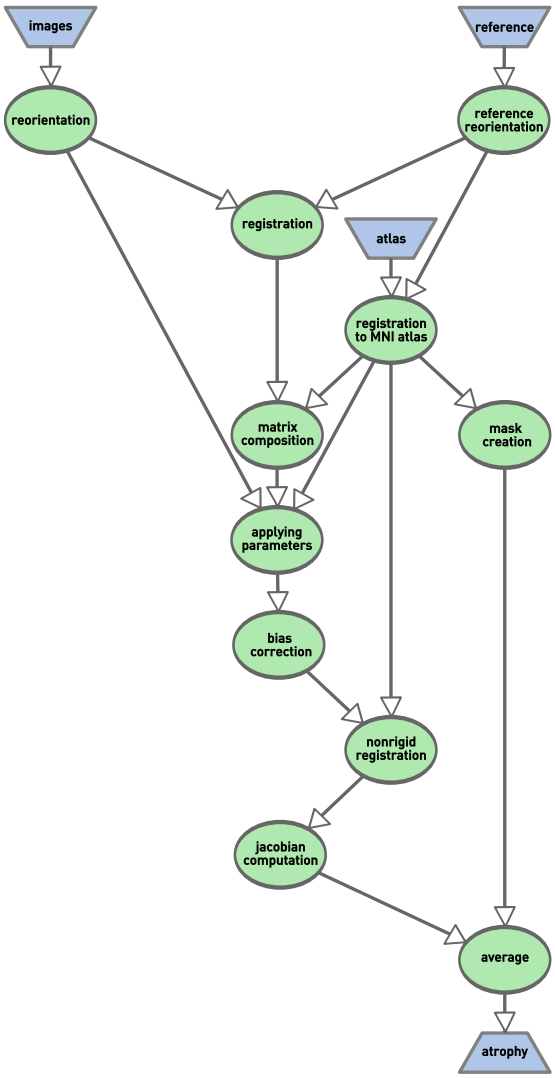


Fig. 2: Simplified scientific workflow schema of the Alzheimer's disease case study where the green ellipses represent services, and the blue trapezoids input/output data.

In terms of execution the longitudinal atrophy detection in Alzheimer's disease is a good example for the validation of the decision model because services composing the workflow are heterogeneous in terms of average execution time and memory consumption as shown in Table I. A benchmark of the average execution time of each service on the target DCI was previously done to estimate the values of t_i, r_i, and T_{MAX}.

Services	Average time [min]	Memory [MB]
images reorientation	1.450	150
reference reorientation	1.450	150
rigid registration	3.217	250
registration to MNI atlas	4.183	250
matrix composition	1.333	150
applying parameters	2.317	200
bias correction	7.167	500
mask creation	14.350	1,000
nonrigid registration	174.783	6,500
Jacobian computation	3.300	1,000
average	1.333	150

TABLE I: Benchmark of average services execution on EGI

B. Experimental setup

Three experiment types were defined to test different submission approaches supported by *jigsaw* and validate the framework scalability:

1) **Execution on grid**. The workflow is executed by submitting jobs directly to the EGI WMS. This is the default behavior when working on production environments and it is considered as baseline performance.
2) **Multilevel scheduling execution**. The workflow is executed using DIRAC. It represents a basic environment considering pilot jobs.
3) **Efficient execution**. The workflow is executed implementing all optimization mechanisms of the framework. Two services containers are deployed to instrument the decision model. One server manages executions using local resources, and the second one serves the job submissions on EGI through DIRAC.

Several patients could be processed in parallel without performance loss assuming availability of resources on the DCI. For each experiment type, the workflow was executed with patients' datasets which size grows exponentially from 1 to 256, and with 2 to 5 images associated to each patient. This leads to an average of 25 service executions per patient submitted to the infrastructure. The experiments were performed on the European Grid Infrastructure, using inputs of the Alzheimer's Disease Neuroimaging Initiative (ADNI) database. For the local executions a server with 2 quad–core processors at 2.67 GHz and 16 GB of memory was used.

We are mainly interested in the final workflow execution timespan for the speedup calculation and the average latency of all job submissions. Nevertheless, the execution failure rate is also reported as it is an important element in the reliability analysis.

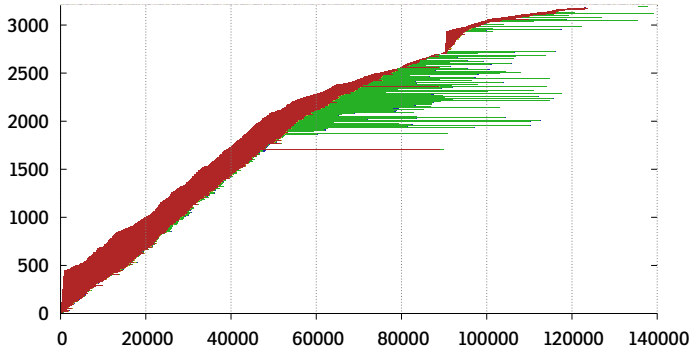
C. Latency

Table II details the statistical results of all three experiment types for the latency analysis including the average latency ($\bar{x}$), the standard deviation (σ), the median absolute deviation (MAD), and the interquartile range (IQR). The MAD and IQR are robust statistics that are not significantly affected by outliers. The variable workload of a production environment like EGI is noticed obtaining higher values of σ in the grid execution type. The robust statistics are preferred for the analysis, in the context of executions on EGI, because outliers have exhibited a high impact on latency due to load variability [4]. Globally, we observe a sustained latency increment when the input dataset size increases. This behavior is expected as the increasing number of jobs loads the submission queues (e.g., from 5 concurrent executions for 1 patient up to 962 for 256 patients). Nevertheless, the introduction of multilevel scheduling reduces significantly the average latency. Focusing on the largest dataset runs, we verify that the multilevel scheduling optimization reduces the latency of 85.25% compared to grid execution for 256 patients. The latency reduction of the efficient execution is equivalent to the multilevel one reaching 86.13%, confirming the benefit effect of pilot jobs in both execution types. This behavior is verified by the MAD, a variability measure comparable to σ. In addition, the MAD exhibits the repercussions of introducing the decision model into the framework for job submission fairness improvement. The values of MAD for efficient executions are reduced to slight or null values in efficient executions compared to the multilevel execution. Similarly, the range of average latency is attenuated significantly. We observe a reduced variability of latency reflected with lower values of IQR. However, the use of limited local resources shows up with large datasets obtaining similar IQR values in multilevel and efficient executions.

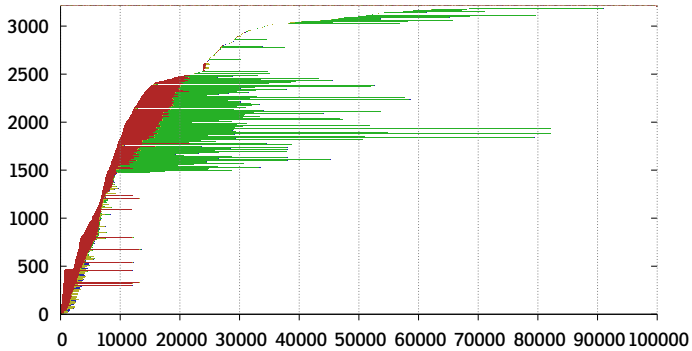
Type	Patients	$\bar{x}$ [min]	σ	MAD	IQR
Grid	1	1.815	1.712	0.433	2.883
	2	2.783	3.180	0.667	3.700
	4	2.871	4.049	0.675	3.284
	8	12.251	13.836	5.208	10.134
	16	35.141	33.672	11.467	28.666
	32	39.841	29.903	10.500	29.850
	64	52.237	145.353	13.583	43.484
	128	107.185	53.747	27.417	78.367
	256	178.289	101.185	51.008	100.467
Multilevel	1	1.525	1.112	0.400	1.066
	2	2.049	2.354	0.504	1.367
	4	3.558	6.031	0.350	2.084
	8	2.428	2.750	0.408	2.267
	16	5.349	10.609	0.880	3.867
	32	10.017	24.844	1.138	5.142
	64	6.637	8.637	1.846	9.825
	128	14.134	17.741	7.896	19.350
	256	26.293	40.736	8.269	32.389
Efficient	1	0.304	1.019	0.000	0.000
	2	0.582	1.954	0.000	0.000
	4	0.477	1.349	0.000	0.000
	8	0.460	1.152	0.000	0.000
	16	1.559	5.174	0.000	0.950
	32	5.470	17.212	0.171	3.463
	64	6.205	11.644	0.900	6.000
	128	10.279	16.120	1.517	15.350
	256	24.730	48.920	2.900	29.217

TABLE II: Statistics summary for submission latency: average latency over 3 executions ($\bar{x}$), standard deviation (σ), median absolute deviation (MAD), and interquartile range (IQR).

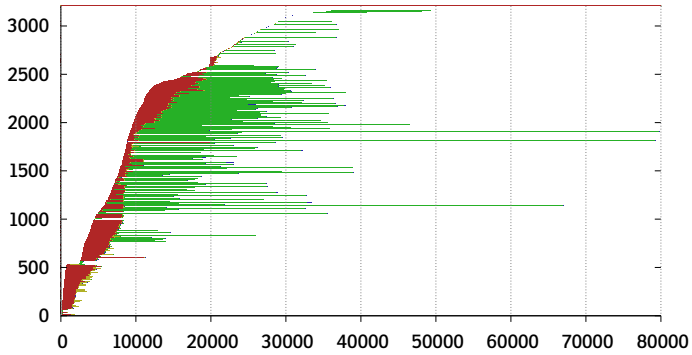
In Fig. 3, we present the timeline diagrams for the execution of the three identified experiment types. Graphically, we observe the time evolution of a saturated workload manager during the *grid execution* that results in all tasks having a similar waiting time. This latency is gradually reduced during the *multilevel scheduling execution*. Finally, we observe a



(a) Grid



(b) Multilevel



(c) Efficient

Fig. 3: Execution timeline charts (number of services as a function of the timespan in seconds) for the workflow executions of 128 patients on EGI. Each service execution is represented by a horizontal bar composed of two parts: the first (in red) is the latency before obtaining computing resource, the second (in green) is the execution time including data transfers.

scaled latency attenuation effect when local resources are used in the *efficient execution* type. However, the dynamic resource acquisition is exhibited once the pilot jobs are enabled in multilevel and efficient executions.

D. Speedup

Two types of speedup are considered to evaluate the impact of the execution framework on the application performance: the *traditional speedup* and the *workflow speedup*. The traditional speedup S is defined as the ratio of a reference, sequential running time of the application over the timespan measured during a parallel run. This value assesses the interest of using DCIs for executing large–scale applications instead of running applications sequentially. It varies with the parallel execution mode considered and the input data set processed. On the other hand, we determine the workflow speedup $S_w = p \times T_1/T_p$, where p is the number of patients, and T_i is the execution time for i patients in a given execution type. In practice, we obtain the value of S_w with regard a constant reference time T_1 of the *grid execution*. Acting this way, S_w represents a good comparator between execution types.

Table III presents the statistical results of the timespan, failure rate, and the computed speedups for each execution type. For all execution, the timespan grows when the size of input dataset increases. However, an exception is verified in case of multilevel scheduling execution of 128 patients, which is longer than the 256–patient run. This is due to its high failure rate leading to the resubmission of 9.09% of total number of tasks. This behavior exhibits the dynamic workload of production Grid infrastructures.

Concerning speedups, we observe that they are effective from one patient ($S \geqslant 1$). The increasing speedup verifies the several levels of parallelism (data, service, pipeline) implemented in the workflow enactment system. The speedup increases significantly even if the latency increases showing the relevance of the resources availability on DCIs. The workflow enactment enables concurrent executions improving the final execution timespan specially for large number of patients. In the best case of efficient execution, the traditional speedup reaches a factor of 120.915 in comparison to the sequential execution. When comparing S_w of the multilevel scheduling execution to the grid one, we observe that pilot jobs reduce significantly the execution timespan. The workflow speedup attains a factor of 102.918 for the largest input dataset.

Combining the local resources to the execution framework lightly reduces the final execution timespan. It confirms that the use of local resources does not reduce the final execution timespan significantly, but it has a clear influence on latency and reliability as expected in a high throughput environment. Therefore, S_w of the efficient execution is relatively the same as the multilevel one.

E. Reliability

We observe in Table III a significant failure rate within each execution type. These rates are influenced by several factors on the production environment, namely full storage elements, temporal unavailability of middleware services such as the file catalog server or the proxy certificates manager, unexpected timeouts while storing data, or application specific errors due to incompatibilities with OS computing elements and/or missing system libraries. For instance, the failure rate of the default grid execution type is high, up to 31.86% in the worst case. The reduction of the failure rate in the multilevel scheduling execution (up to 13.71% in the worst case) is possible thanks to the sanity check mechanism of pilot jobs, and broken computing resources filtering before jobs pulling. When executing jobs to local resources, we also eliminate the errors concerning the incompatibilities with OS computing elements and/or missing system libraries. The failure rate

Type	Patients	Timespan [hours]	Failure rate	S	S_w
Grid	1	8.024	0.00%	1.749	1.000
	2	6.556	25.00%	3.058	2.448
	4	7.326	14.29%	4.047	4.381
	8	14.394	31.86%	5.812	4.460
	16	21.144	17.46%	10.567	6.072
	32	22.442	27.77%	15.979	11.441
	64	33.619	11.31%	17.020	15.275
	128	35.863	14.83%	37.051	28.639
	256	41.531	11.36%	57.500	49.460
Multilevel	1	3.382	0.00%	3.439	2.373
	2	4.569	10.26%	4.694	3.512
	4	4.484	1.82%	8.262	7.158
	8	4.478	2.26%	15.488	14.335
	16	6.200	1.51%	16.800	20.706
	32	8.614	2.02%	26.407	29.808
	64	12.831	13.71%	54.423	40.023
	128	20.528	9.09%	56.527	50.033
	256	19.959	1.99%	93.043	102.918
Efficient	1	3.152	0.00%	3.222	2.546
	2	3.574	0.00%	5.670	4.490
	4	3.461	0.00%	9.249	9.274
	8	3.354	0.46%	16.737	19.139
	16	4.048	0.92%	28.445	31.715
	32	7.750	1.02%	28.288	33.131
	64	9.560	6.13%	40.610	53.717
	128	12.536	6.95%	76.742	81.930
	256	18.655	7.42%	120.915	110.112

TABLE III: Statistics summary for timespan and speedup

therefore reduces up to 7.42% in the worst case. It means that the use of local resources does not represent a error safeguard, specially in case of executions with large datasets becoming only a improving method to attenuate failure rates and latency.

VI. Conclusion

In spite of more than a decade spent in active research and development of DCIs middleware, large–scale infrastructures often remain accessible to experts only. The inherent complexity of such systems is causing large overheads, numerous failures, and complex application control environment. This paper proposed pragmatic solutions to deploy and control execution of large–scale applications. The solution proposed is based on state–of–the–art middleware components, glued together in a coherent service–oriented environment that interfaces these components so as to balance the features provided by each of them. As a result, complex scientific experiments such as the study on Alzheimer's exemplified in this paper can be reliably enacted and high performing. The computing environment was built from well–established software component in a general setting to make it as reusable as possible.

Acknowledgments

This work is supported by the SHIWA project (http://www.shiwa-workflow.eu) under a grant from the European Commission's FP7 INFRASTRUCTURES-2010-2 call, agreement number 261585. The authors would like to thank the Asclepios Research Project from Inria Sophia–Antipolis for providing the case study used in this work.

References

[1] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, "Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests," in *HPDC–11*, Edinburgh, UK, Jul. 2002.

[2] H. Casanova, "Benefits and Drawbacks of Redundant Batch Requests," *Journal of Grid Computing*, vol. 2, no. 5, 2007.

[3] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *SOSP'09*, Big Sky (MT), Oct. 2009.

[4] D. Lingrand, J. Montagnat, and T. Glatard, "Modeling User Submission Strategies on Production Grids," in *HPDC'09*, Munich, Germany, Jun. 2009.

[5] C. Dabrowski, "Reliability in Grid Computing Systems," *Concurr. Comp.-Pract. E.*, vol. 21, no. 8, 2009.

[6] E. Huedo, R. S. Montero, and I. M. Llorente, "Evaluating the Reliability of Computational Grids from the End User's Point of View," *J. Syst. Architect.*, vol. 52, no. 12, 2006.

[7] A. Kangarlou, P. Eugster, and D. Xu, "VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime," in *DSN'09*, Lisbon, Portugal, Jun. 2009.

[8] G. Koslovski, W.-L. Yeow, C. Westphal, T. Truong Huu, J. Montagnat, and P. Vicat-Blanc Primet, "Reliablility Support in Virtual Infrastructures," in *CloudCom 2010*, Indianapolis (IN), Nov. 2010.

[9] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev, and the Lhcb Dirac Team, "DIRAC Pilot Framework and the DIRAC Workload Management System," *J. Phys. Conf. Ser.*, vol. 219, no. 1–6, 2010.

[10] S. Krishnan and K. Bhatia, "SOAs for Scientific Applications: Experiences and Challenges," *Future Gener. Comp. Sy.*, vol. 25, no. 4, 2009.

[11] R. Ferreira da Silva, S. Camarasu-Pop, B. Grenier, V. Hamar, D. Manset, J. Montagnat, J. Revillard, J. Rojas Balderrama, A. Tsaregorodtsev, and T. Glatard, "Multi–infrastructure Workflow Execution for Medical Simulation in the Virtual Imaging Platform," in *HealthGrid 2011*, Bristol, UK, Jun. 2011.

[12] I. D. Dinov, J. D. Van Horn, K. M. Lozev, R. Magsipoc, P. Petrosyan, Z. Liu, A. MacKenzie-Graham, P. Eggert, D. S. Parker, and A. W. Toga, "Efficient, Distributed and Interactive Neuroimaging Data Analysis Using the LONI Pipeline," *Front. Neuroinform.*, vol. 3, 2009.

[13] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk, "GEMLCA: Running Legacy Code Applications as Grid services," *Journal of Grid Computing*, vol. 3, no. 1, 2005.

[14] K. Chard, W. Tan, J. Boverhof, R. Madduri, and I. Foster, "Wrap Scientific Applications as WSRF Grid Services Using gRAVI," in *ICWS'09*, Los Angeles (CA), Jul. 2009.

[15] I. Foster, "Globus Toolkit Version 4: Software for Service–oriented Systems," *J. Comput. Sci. & Technol.*, vol. 21, no. 4, 2006.

[16] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *Services 2007*, Salt Lake City (UT), Jul. 2007.

[17] T. Oinn, M. Greenwood, M. J. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences," *Concurr. Comp.-Pract. E.*, vol. 18, no. 10, 2006.

[18] I. Taylor, I. Wand, M. Shields, and S. Majithia, "Distributed Computing with Triana on the Grid," *Concurr. Comp.-Pract. E.*, vol. 17, no. 9, 2005.

[19] J. Rojas Balderrama, J. Montagnat, and D. Lingrand, "jGASW: A Service-oriented Framework Supporting HTC and Non-functional Concerns," in *ICWS 2010*, Miami (FL), Jul. 2010.

[20] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and Efficient Workflow Deployment of Data–intensive Applications on Grids with Moteur," *Int. J. High. Perform. Comput. Appl.*, vol. 22, no. 3, 2008.

[21] The Apache Software Foundation, "Apache Tomcat Web Server and Servlet Container," http://tomcat.apache.org.

[22] D. Kouril and J. Basney, "A Credential Renewal Service for Long-Running Jobs," in *Grid 2005*, Seattle (WA), Nov. 2005.

[23] S. D. Olabarriaga, T. Glatard, and P. T. de Boer, "A Virtual Laboratory for Medical Image Analysis," *IEEE T. Inf. Technol. B.*, vol. 14, no. 4, 2010.

[24] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. Blay-Fornarino, "A Data-driven Workflow Language for Grids Based on Array Programming Principles," in *WORKS'09*, Portland (OR), Nov. 2009.

[25] M. Lorenzi, X. Pennec, and G. B. Frisoni, "Monitoring The Brain's Longitudinal Changes in Clinical Trials for Alzheimers Disease: A Robust and Reliable Non–rigid Registration Framework," in *AAIC'11*, Paris, France, Jul. 2011.